Development of a Software Security Assessment Instrument to Reduce Software Security Risk

David P. Gilliam
Caltech, Jet Propulsion Laboratory
david.p.Gilliam@jpl.nasa.gov

John C. Kelly
Caltech, Jet Propulsion Laboratory
john.c.kellyw@jpl.nasa.gov

John D. Powell
Caltech, Jet Propulsion Laboratory
John.Powell@jpl.nasa.gov

Matt Bishop
University of California at Davis
bishop@cs.ucdavis.edu

Abstract

This paper discusses joint work by the California Institute of Technology's Jet Propulsion Laboratory and the University of California at Davis (UC Davis) sponsored by the National Aeronautics and Space Administration to develop a security assessment instrument for the software development and maintenance life cycle. The assessment instrument is a collection of tools and procedures to support development of secure software.

The security assessment instrument includes a Vulnerability Matrix (VMatrix) with platform/application, and signature fields in a database keyed on the Common Vulnerabilities and Exposures (CVE) number. The information in the VMatrix has become the bases for the Database of Vulnerabilities, Exploits, and Signatures (DOVES) at UC Davis. The instrument also includes a property-based testing tool to slice software code looking for specific vulnerability properties. A third component of the research is an investigation into the verification of software designs for compliance to security properties. This is based on innovative model checking approaches that will facilitate the development and verification of software security models

Keywords

Security Toolset, Vulnerability Matrix, Property-Based Testing, Model Checking, Security Verification

1. Introduction

Software on networked computer systems must be free from security vulnerabilities. Security vulnerabilities in software arise from a number of programming factors, but which can generally be traced to poor software development practices, new modes of attacks, mis-configurations, and unsecured links between systems. An otherwise secure system can be compromised easily if a system or application software on it, or on a linked system, has vulnerabilities.

Currently, there is a lack of SATs for use in the software development and maintenance life cycle to mitigate these vulnerabilities. The National Aeronautics and Space Administration (NASA) has funded the Jet Propulsion Lab in conjunction with the University of California at Davis (UC Davis) to develop a software security assessment for use in the software development and maintenance life cycle.

The goal of the effort is the use of a formal analytical

approach for integrating security into existing and emerging techniques for developing high quality software and computer systems. The approach is multifaceted, with activities and prototype tools in the following sub-domains:

- Assessment instrument for reducing risk during development, configuration, and installation of secure systems
- Model based development and verification for secure software architectures
- Security testing, and verification and validation (V&V) techniques

Assessments of high profile NASA systems believed to be vulnerable to attack will provide a metric to determine the effectiveness of these activities and prototypes.

The inception of this work was previously reported to IEEE WETICE Workshop on Enterprise Security.[1] Two parts have been accomplished to date, the Vulnerability Matrix (Vmatrix) and the Security Assessment Tools (SATs). A third part, the property-based testing Tester's Assistant (TA), will be completed in June. The security assessment instrument will be tested on a JPL/NASA Class A Flight Project to verify the approach and the viability of the security assessment instrument for assuring the security of software on critical networked systems.

2. Vulnerability Matrix (Vmatrix)

The VMatrix task was initiated to develop a searchable database containing a taxonomy of vulnerabilities and exposures. The information in the database is intended, in part, to provide network security professionals an understanding of the vulnerabilities and their exploits so they can better secure their systems. Equally important, it also provides developers with an understanding of the vulnerabilities and exposures in code that introduce security risks to software and systems. The intended goal is to enable developers to write more secure code and to model and test it to mitigate these security risks.

The Vmatrix, examines vulnerabilities and exposures and the methods used to exploit them. The VMatrix lists vulnerabilities and exposures along with their Common Vulnerabilities and Exposures (CVE) listing[2]. The VMatrix includes a brief summary and a description of the vulnerability or exposure, the affected software or operating system, how to detect the vulnerability or exposure and the fix or method for protecting against the exploit. Also included is catalogue information, keywords, and other related information as available, regarding the vulnerability or exposure. Interesting links, including links to Mitre with the CVE listing and the Ernst and Young website where vulnerabilities and exposures are ranked by severity and frequency among other factors, are also provided.

The VMatrix led to the development and extension of a database controlled and maintained by UC Davis, the Database of Vulnerabilities, Exploits, and Signatures, (DOVES). DOVES contains additional vulnerabilities and exposures beyond that which is now contained in the VMatrix.

3. Security Assessment Tools (SATs)

The SATs are a collection of security assessment and testing tools to evaluate systems and the software code running on them. Each SAT has a brief summary stating the purpose of the tool and its use along with pros and cons of the tool. Also provided is a list of similar tools or alternative tools, and a classification of the tool. Additionally, the discussion of each tool includes a website where the tool can be found. A journal paper, "A Classification Scheme for Security Tools," provided on the SATs web page, discusses a classification scheme of these security related tools and their usage.

A more complete description of the tools and a discussion of how to use each of the tools is currently being developed. Additional SATs are being collected as they become available to include in the current list.

The SATs will be categorized and cross referenced to alternate tools so that code developers, system administrators, and network and computer security professionals can have a central location to search for specific tools for use in writing secure software code and securing computer systems.

4. Property-Based Testing

The role of property-based testing is to bridge the gap between formal verification and *ad hoc* verification. This provides a basis for analyzing software without sacrificing usefulness for rigor, yet capturing the essential ideas of formal verification. It also allows a security model to guide the testing for security problems

This work was previously reported at last year's WETICE, June 2000. During development of the tool the direction has slightly changed. The original work specified that we would develop the TA property-based testing tool to test properties in C++ code. However, the TA task has been refocused to test for security properties in JAVA instead.

Property-based testing [2] is a technique for testing that programs meet given specifications. The tester gives the specifications in a language that ties the specification to particular segments of code. The specification has *assertions*, which indicate changes in the security state of

the program, and *properties*, which describe a specific security state (that, in this context, is considered secure). The idea is to ensure that the properties always hold.

To simplify the testing procedure, the program is *sliced* to delete those parts of the program unrelated to the properties being tested. The result of the slicing is a compilable program that satisfies the properties if, and only if, the unsliced program satisfies the properties.

The sliced program is then modified to function as a simple state machine by inserting code to emit information about assertions and the need to check properties. A monitor collects this information and updates the state of the program. When a property statement is reached, the monitor determines if the property holds. If not, the monitor reports that the program failed to meet its specification, and the tester can take appropriate action.

Next, the program is sliced, or the smallest program equivalent to the original with respect to the stated properties is derived. Then the program is instrumented and tested, as described earlier. The testing either validates the properties or shows they do not hold. This helps determine the level of assurance of the program, as captured by the arrow going from the right to the left.

As an example, consider Fink's implementation of property-based testing. He defined a language called TASPEC that expressed specifications in terms of C code (which was the environment in which his tool was to be used). This differs from more familiar specification languages such as Z, which work at a more abstract level. For example, consider the high-level requirement (stated in English) that "a user must authenticate himself or herself before acquiring privileges." The low-level specification (again in English) for a UNIX program written in C would be:

```
Is password corrrect? {
        Compare user's password hash to hash stored
                for that user name
        If match, set UID to user's uid
        If no match, set UID to ERROR
}
if privileges granted {
        compare UID to the uid for which privileges
                are granted
        if match, all is well
        if no match, specification violated
}
```

Translating this into TASPEC gives [3]:

```
location func setuid(uid) result 1
        { assert privileges_acquired(uid); }
location func crypt(password,salt) result encryptpwd
        { assert password_entered(encryptpwd); }
location func getpwnam(name) result pwent
        { assert user_password(name,
            pwent->pw_passwd, pwent->pw_uid); }
location func strcmp(s1, s2) result 0
        { assert equals(s1, s2); }
password_entered(pwd1) and
        user_password(name, pwd2, uid) and
            equal(pwd1, pwd2)
        { assert authenticated(uid) ; }
authenticated(uid) before privileges_acquired(uid)
```

The set of statements with **assert** in their associated blocks are the set of TASPEC statements that indicate changes of state. From the top, they say that when the program makes a *setuid* system call with an argument of *uid*, the process acquires the privileges of that *uid*; when the process calls the UNIX function *crypt*, the result is a hashed password; when the process calls the *getpwnam* function, it gets back information about the user, her (hashed) password, and UID; and the UNIX function *strcmp*, when called, compares two strings and returns 0 if they are equal. Code is added to the named functions so that, for example, when *setuid* is called, the assertion *privileges_acquired(uid)* is added to the current security state. When all of *password_entered(pwd1)*, *user_password(name, pwd2, uid)*, and *equal(pwd1, pwd2)* are in the state, the monitor will add *authenticated(uid)* to the state.

The last line says that when the assertion *setuid(uid)* is added to the state, the property that *authenticated(uid)* is already in the state, for the same *uid*, must hold; in other words, if *authenticated(uid)* is not in the current security state, the monitor will raise a warning that a property has not been satisfied.

This example shows how property-based testing can take advantage of the specifications of vulnerabilities in VMatrix to detect problems. The key is to represent the vulnerability in the low-level specification language. That explains the subfields in the matrix; they must capture the essence of the problem. As a side benefit, this work feeds directly into the Davis model for vulnerabilities [4], because the assertions are the preconditions of that model.

A second advantage of the properties being stored in VMatrix is that they form the core of a library that can be used for testing the security of programs other than those in the NASA suite. In essence, we extend the notion of software reuse to properties and assertions [5].

Currently, the execution monitor has been implemented. This PROLOG tool verifies that the state of the program or system being monitored is consistent with assertions describing the desired security properties. In particular, several ambiguities of the language were resolved (for example, in the expression "x until y", must property x hold from the start, and then when property y becomes true, must property x become false or can it remain true?) and some operations added (such as

"assertonece(x)", which deletes the assertion "x" immediately after it is verified; this enables one-time properties to be checked and retracted immediately). The instrumenting program has been retargeted to Java, and as of now the parser and some instrumenting code are completed.

We are applying the property-based testing tool to the NASA environment. Hence, the prototype property-based testing tool is being written initially for JAVA and the UNIX environment. Success will be determined by performing static analysis of several programs of interest. Flaws not known to those performing the testing will be injected into the program to provide a baseline of measurement. Once this is completed, we will develop and prototype an engine to perform testing on programs in a production type environment.

At this point we examine the use of a run-time testing tool similar to the static property-based testing tool (the difference being the omission of slicing). We determine how much its use degrades performance, and how much extra overhead the testing adds to the system in general. We also perform "friendly" penetration attacks on the relevant software, and determine if the run-time testing environment detects these attacks .The property-based software tool, TA–Next Generation (TANG), will benefit users by giving them increased confidence in the programs' correctness with respect to the stated (security) specifications. If non-security properties (such as safety) are of interest, the testers will have access to TANG to perform similar testing for their own set of properties. But the focus of this study is on security-related properties.

5. Model-Based Security Specification and Verification

Analyses based on discrete finite models can be used to verify and check compliance to desired security properties. Many security properties cannot be verified by test activity alone, however verification through analyses and modeling at the design stage can increase the confidence that the specification provides a sound base for developing a secure program or communication protocol. The analysis and modeling process can begin early in the software development life cycle. Modeling tools and languages used together provide a machine-readable model that facilitates automated verification of system properties. Models should be updated periodically, as requirements and designs become more mature. Analysis of up-to-date models can contribute to verification by testing programming code through test case generation via Model Checking. [6,7]

Software model checkers automatically explore all paths from a start state in a computational tree. The computational tree may contain repeated copies of sub-trees. State of the art Model Checkers such as SPIN exploit this characteristic to improve automated verification efficiency. The objective is to verify system properties with respect to models over as many scenarios as feasible. Since the models are a selective representation of functional capabilities under analysis, the number of feasible scenarios is much larger than the set that can be checked during testing. Model Checkers differ from traditional formal techniques by the following characteristics:

- Model checkers are operational as opposed to deductive
- Model checkers provide counter examples when properties are violated (error traces)
- Their goal is oriented toward finding errors as opposed to proving correctness since the model is an abstraction of the actual system

Model based security specification and verification:

Model checking addresses issues in security protocols by examining a large number of ways to circumvent the security mechanism. In contrast to purely analytic methods, model checking is capable of examining the larger venue by validation of the overall security system in local, regional, or global environments. These methods have more leverage since they model real world scenarios, and they embrace more than just the mathematics of the protocol. For example, the Needham-Schroder protocol (1978) was proven secure using the BAN logic for protocol specification. However, Lowe (1998) and Wu (1998) using the model checking system SPIN, have discovered successful attacks abrogating the effectiveness and usefulness of this protocol.[8,9] We propose to extend this approach to protocol validation by (1) proposing models of security protocol systems, and (2) validating those configurations. These modeling techniques have developed around a multi-agent programming paradigm that has emerged as a convenient framework around which internet applications can be successfully validated.

However, Model Checking suffers from the known drawback of "State Space Explosion". The state space that a Model Checker must exhaustively explore grows at the rate of m^n where:

- n is the number of variables contained in the discrete mathematical model
- m is the range of discrete values that a variable may have

Thus, the model must necessarily be an abstraction of the actual system to make Model Checking feasible with reasonable computing resources. In most cases, any

substantial system will produce an intractable state space if modeled in its entirety. Therefore, a careful choice, based on domain and Model Checking expertise, must be made with regard to the portion(s) of the system to model.

Three common properties to check for are:

Invariant – always p

- p is a property the model must always have

Safety – not ever q

- q is a property the model must never have

Liveness – r implies s will eventually be "true" at some point now or in the future

- always the case that if property r holds at the current state, then property s will hold at some state now or in the future
- used to guarantee that significant sequences take place

Security verification of new architectures:

Architectures that support change and facilitate maintenance are essential to secure systems. However, these architectures are inadequately tested by traditional verification techniques. Model Checking offers ways to begin modeling and investigating the behavior of the planned system, and to validate that key properties hold invariantly in the system as modeled. This technique will be explored in collaboration with security vulnerabilities and property based testing as part of this study.

The modeling of network security systems (NSSs) and validation of the associated properties using Model Checking represents an ambitious undertaking. The complexity of such a system produces state space explosion beyond the capability of state of the art model checkers. This complexity exists on two levels:

- System Complexity - The NSS itself
- Environmental Complexity - The diverse environment (the internet) in which it must operate

Using abstraction as a means of coping with these complexities involves the removal of irrelevant details based on valid assumptions about the system's/environment's behavior. Readily available domain and model checking expertise offers the opportunity for valid abstraction to cope with system complexity. However, the environmental complexity is much higher than the system complexity. Further, the environment is continuously changing and evolving with the constant emergence of new network security attacks. The dynamic nature of the environment is a significant barrier to abstraction because the validity of assumptions may change as the environment changes.

The proposed approach to overcoming the environmental complexity is compositional in nature. A taxonomy of possible atomic network activities will be developed. The environmental model will be divided into independent components based on this taxonomy. The component relationships will be preserved through the Flexible Modeling Framework (FMF) that is being developed for this project. These components will be allowed to act on the NSS in all possible variations via the FMF. Finally a subset of these components and their relationships, deemed valid by the FMF, will be combined with the NSS model to form a maximal sub-model that is within the operational limits of a model checker over its available resources. The FMF preserves the properties within the sub-model such that a verification resulting in a property violation may be validly extrapolated to the full model at large. This facilitates the partial verification of models that are too large for current state of the art Model Checkers.

Partial verification of the model refers to the fact that only properties that do not require the behaviors of the full model (or a very large sub-model) can be fully verified through a sub-model. This is not a new notion. However, the contribution of the FMF is that the modeling by components methodology facilitates automated support for sub-model construction and possible component reuse. Devising a valid sub-model with respect to a given property manually is a tedious process requiring a great deal of domain and model checking knowledge. This approach allows all sub-models whose state space is within the capability of the Model Checker to be examined automatically. The sub-model is constructed on the fly. The property is verified over the sub-model. Then the sub-model is destroyed and a new sub-model is created. The sub models are examined in a smallest to largest sequence. When an error is discovered, the ability to extrapolate the error result will eliminate, the need to examine remaining sub-models because. Note, if no property violation is found via the examination of all sufficiently small sub-models the notion that the property holds for the entire model may not be extrapolated. The ability to extrapolate property violations only is analogous the fact that Model Checking results over a full model can find actual system errors but cannot prove actual system correctness.

6. Conclusion

The four parts of this work form a coherent technique for examining new and existing systems and software code for security flaws. Each part can be used independently or in conjunction with another. When used in conjunction with each other, each part leverages cumulative benefits to classify and focus upon security properties that will be modeled and tested. The VMatrix and model-based checking provide the properties that the software must meet; the property-based tester checks that the programs do indeed meet these properties. The VMatrix forms the

beginning of a library of TASPEC properties. Property-based testing requires properties expressed in TASPEC to test against. Training in the writing of more secure programs flows directly from the library of security properties and the system-specific models. Placing these in the context of a particular language and environment is an important part of improving the quality of software and systems.

7. Acknowledgements

The research described in this paper is being carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration, and the University of California at Davis under a subcontract with the Jet Propulsion Laboratory, California Institute of Technology.

8. References

[1] D. Gilliam, J. Kelly, M. Bishop, "Reducing Software Security Risk Through an Integrated Approach," Proc. of the Ninth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (June, 2000), Gaithersburg, MD, pp.141-146.

[2] Published and maintained by Mitre. The CVE listing can be found at: http://cve.mitre.org/

[3] G. Fink, M. Bishop, "Property Based Testing: A New Approach to Testing for Assurance," *ACM SIGSOFT Software Engineering Notes* **22**(4) (July 1997).

[4] M. Bishop, "Vulnerabilities Analysis," *Proceedings of the Recent Advances in Intrusion Detection* (Sep. 1999).

[5] J. Dodson, "Specification and Classification of Generic Security Flaws for the Tester's Assistant Library," M.S. Thesis, Department of Computer Science, University of California at Davis, Davis CA (June 1996).

[6] J. R. Callahan, S. M. Easterbrook and T. L. Montgomery, "Generating Test Oracles via Model Checking," NASA/WVU Software Research Lab, Fairmont, WV, Technical Report # NASA-IVV-98-015, 1998.

[7] P. E. Ammann, P. E. Black and W. Majurski. "Using Model Checking to Generate Test Specifications," 2nd International Conference on Formal Engineering Methods (1998) pp. 46-54.

[8]G. Lowe. Breaking and Fixing the Needham-Schroeder Public Key Protocol Using CSP and FDR. In TACAS96, 1996.

[9] W. Wen and F Mizoguchi. Model checking Security Protocols: A Case Study Using SPIN, IMC Technical Report, November, 1998.